COMP 322: Fundamentals of Parallel Programming

Lecture 10: Critical sections, Isolated statement

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today's Lecture

- Lecture 10 handout

Introduction

- For the programming constructs async, finish, future, get, forall, the following situation was defined to be a data race error
  - when two accesses on the same shared location can potentially execute in parallel such that at least one access is a write.
- However, there are many cases in practice when two tasks may legitimately need to perform conflicting accesses to shared locations.

Example of two tasks performing conflicting accesses

```
class DoublyLinkedList {
  DoublyLinkedList prev, next;
  . . .
  void delete() {
    isolated { // start of mutual exclusion region (critical section)
      if (this.prev != null) this.prev.next = this.next;
      if (this.next != null) this.next.prev = this.prev
    } // end of mutual exclusion region (critical section)
    . . .
  }
  . . .
}
. . .
static void deleteTwoNodes(DoublyLinkedList L) {
  finish {
    async L.delete();
    async L.next.delete();
  }
}
```

How to enforce mutual exclusion?

- The predominant approach to ensure mutual exclusion proposed many years ago is to enclose the code region in a *critical section*.
    - "In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore."

HJ isolated statement

isolated <body>

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - Two instances of isolated statements, ⟨stmt1⟩ and ⟨stmt2⟩, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.
  - ➔Weak isolation guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested (redundant)
- Isolated statements must not contain any other parallel statement: async, finish, get, forall
- In case of exception, all updates performed by <body> before throwing the exception will be observable after exiting <body>

How small or big should an isolated statement be?

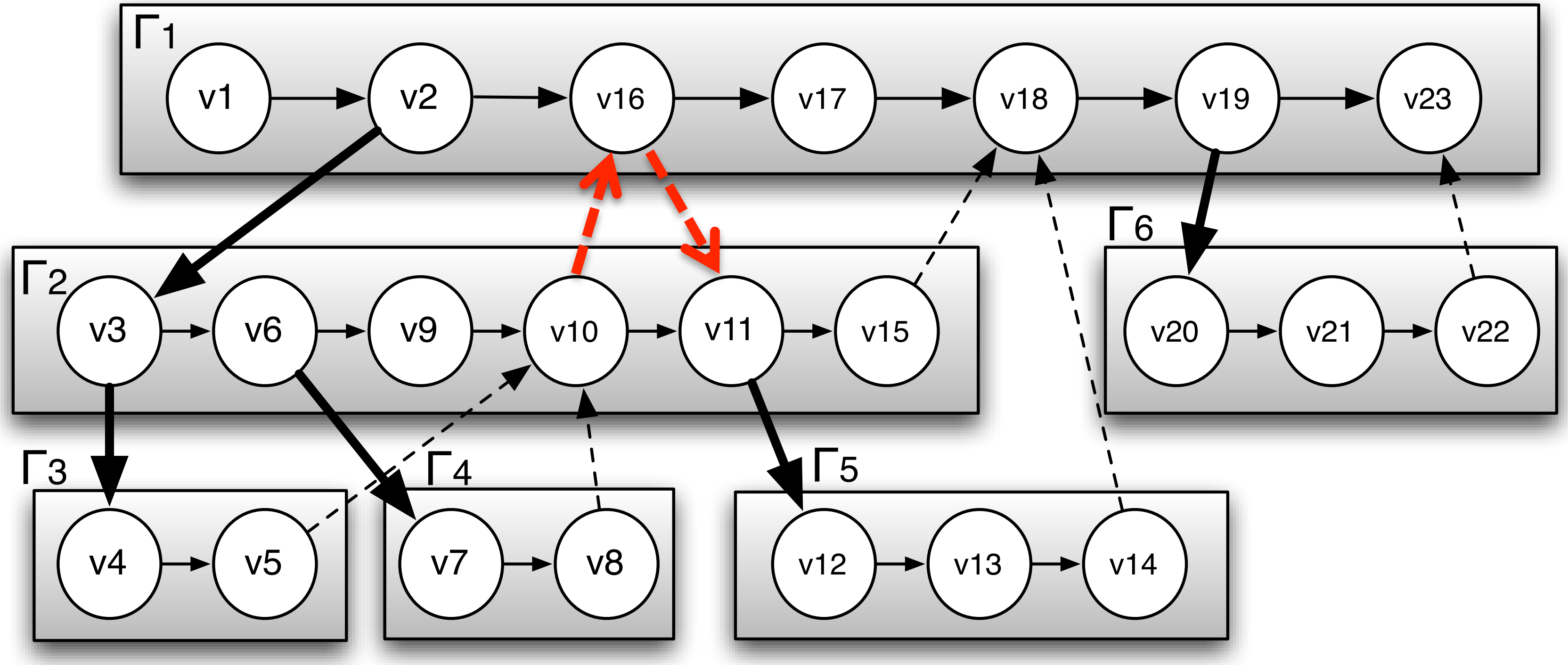
- Too small ➔ may lose invariants desired from mutual exclusion
- Too big ➔ limits parallelism

- Observation: no combination of finish, async, get, forall and isolated constructs can create a deadlock cycle among tasks.

Serialized Computation Graph for Isolated Statements

- Model each instance of an isolated statement as a distinct step (node) in the CG.
- Need to reason about the order in which interfering isolated statements are executed
  - complicated because the order may vary from execution to execution
- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated statements.
  - SCG consists of a CG with additional serialization edges.
  - Each time an isolated step, S', is executed, we add a serialization edge from S to S' for each isolated step, S, that has already executed such that S and S' have interfering accesses.
  - An SCG represents a set of executions in which all interfering isolated statements execute in the same order.

Example of Serialized Computation Graph with Serialization Edges



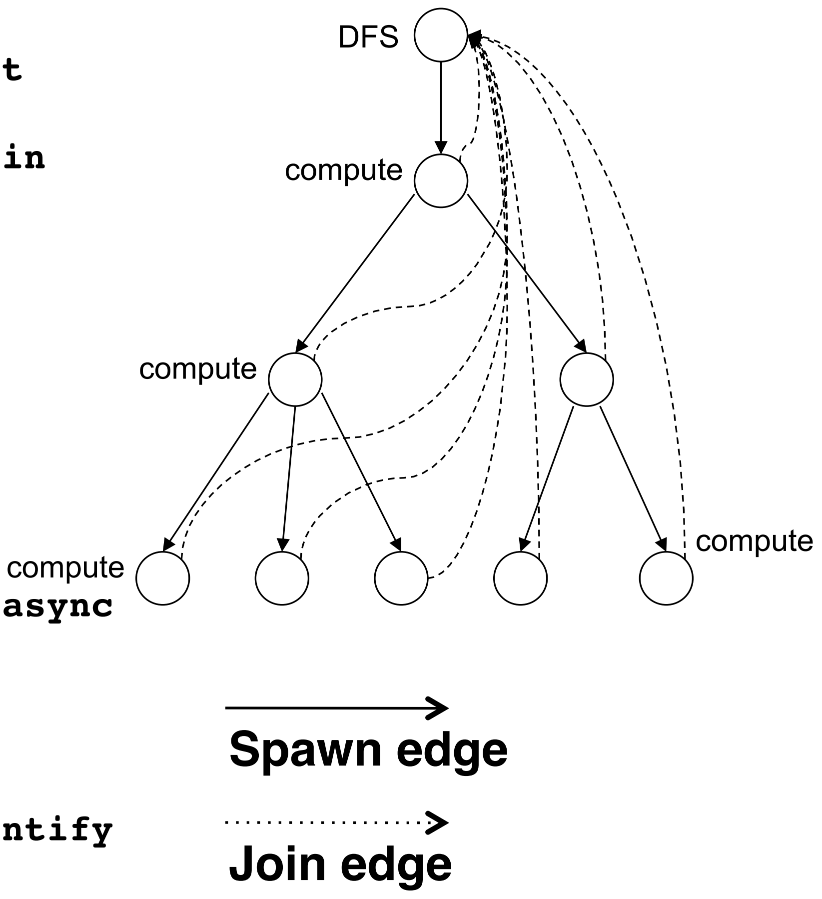
→ Continue edge ➞ Spawn edge - - - - - -▸ Join edge

- - - -> **Serialization edge**

v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

Parallel Depth-First Search Spanning Tree revisited with Object-based Isolation

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent;       // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent=n;
    return parent == n;
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```



Formal Definition of Data Races

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps S1 and S2 in computation graph CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1 i.e., there is no path of dependence edges from S1 to S2 or from S2 to S1 in CG, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

Apply above definition to an SCG